Jadavpur University

Department of Computer Science and Engineering

NETWORKS LAB ASSIGNMENT 3

BCSE UG-III

Student : Arjeesh Palai

Roll No. : 002310501086

Group : A3

Date : 13 / 10 / 2025

Assignment Brief

Course: CSE/PC/B/S/314 Computer Networks Lab **CO3:** Design and implement medium access control mechanisms within a simulated network environment using IEEE 802 standards.

Task: Implement a p-persistent CSMA technique with collision detection (CSMA/CD). Measure and compare:

- **Throughput**: average number of payload data bits delivered per second.
- **Forwarding delay**: end-to-end delay (queuing + transmission + access).
- **Efficiency**: normalized throughput vs. p.

Use the same sender–receiver design as previous assignments. Add a sender-side *Collision Injection* module to stress CSMA/CD. Vary p and report observations.

1 Design

Topology. Multiple client senders connect to a receiver over TCP loopback (port 5000). The receiver acts as a shared medium with collision detection; clients implement p-persistent carrier sensing, slotting, and binary exponential backoff (BEB).

Frames. IEEE 802.3–style header fields and CRC utilities are provided for compatibility and future extensions.

Metrics. Each successful frame contributes *payload bits* and a *tx finish timestamp*; throughput and average forwarding delay are computed across all clients. Intermediate stats are printed periodically.

Module responsibilities

- **Sender (Client)**: carrier sense; transmit with probability p on an idle slot; optional *collision injection*; wait for ACK or timeout; on failure, BEB and retry; accumulate success bits and per-frame delay.
- **Receiver (Server)**: coarse CD by guarding a shared *channel busy* flag; if a new arrival finds the channel busy, count a collision and drop; otherwise, mark busy, service, and send `ACK`; periodic monitor prints active clients and collisions.

2 Implementation (Key Snippets Only)

`common.h` — 802.3 header + prototypes

```
#pragma once
#include <cstdint>
```

```
#include <string>
#include <vector>

extern const uint8_t SENDER_ADDR[6];
extern const uint8_t RECEIVER_ADDR[6];

struct FrameHeader
{
    uint8_t src[6];
    uint8_t dest[6];
    uint16_t length;
    uint8_t seq;
} __attribute__((packed));

...

void fill_header(FrameHeader &h, const uint8_t src[6], const uint8_t
    dest[6],
                 uint16_t length, uint8_t seq);

void append_header(std::vector<uint8_t> &out, const FrameHeader &h);
void append_payload(std::vector<uint8_t> &out, const std::string &p)
   ;
void append_crc(std::vector<uint8_t> &out, Crc32 c);

std::vector<uint8_t> bytes_for_crc(const FrameHeader &h, const std::
   string &payload);

bool read_exact(int fd, void *buf, size_t n);
bool write_exact(int fd, const void *buf, size_t n);

bool is_supported_crc(int widthBits);
```

`common.cpp` — addresses + robust write

```
#include "common.h"
#include <cstring>
#include <arpa/inet.h>
#include <unistd.h>
s
const uint8_t SENDER_ADDR[6]   = {0xAA, 0xBB, 0xCC, 0x11, 0x22, 0x33
   };
```

```
const uint8_t RECEIVER_ADDR[6] = {0xDE, 0xAD, 0xBE, 0xEF, 0x44, 0x55
    };

static uint32_t poly_for(int w)
{
    switch (w)
    {
    case 8:  return 0x07;         // CRC-8
    case 10: return 0x233;        // CRC-10
    case 16: return 0x1021;       // CRC-16-CCITT
    case 32: return 0x04C11DB7;   // CRC-32
...
```

```
bool write_exact(int fd, const void *buf, size_t n)
{
    const uint8_t *p = (const uint8_t *)buf;
    size_t sent = 0;
    while (sent < n)
    {
        ssize_t r = ::write(fd, p + sent, n - sent);
        if (r <= 0)
            return false;
        sent += (size_t)r;
    }
    return true;
}
```

`receiver.cpp` — ACK service + channel-busy reset; summary

```
send(client_sock, "ACK", 3, 0);
{
    lock_guard<mutex> lock(channel_mtx);
    channel_busy_flag = false;
}
...
cout << "\nCSMA/CD p persistence complete\n";
cout << "Total clients: " << client_id << "\n";
cout << "Total collisions detected: " << collision_count.load() << "
    \n";
cout << "Total active time: " << total_time_s << "\n";
append_receiver_csv_row(client_id, collision_count.load(),
    total_time_s);
```

client.cpp — KPI computation & logging

```
double throughput_bps = final_bits / total_time_s;
double avg_delay_ms   = final_frames ? (final_delayus / (double)
   final_frames / 1000.0) : 0.0;

cout << "\nThe Transfer has been Completed\n";
cout << "Number of Clients: " << n_clients << ", Frames per client:
   " << frames_per_client << "\n";
cout << "Throughput (bps): " << throughput_bps << "\n";
cout << "Avg fwding delay: " << avg_delay_ms << "\n";

append_client_csv_row(
    n_clients, frames_per_client, P_persistent, slot_ms,
    ack_timeout_ms, max_BEB_k, collisionProb,
    final_frames, final_bits, total_time_s, throughput_bps,
       avg_delay_ms);
```

3 Build & Run (sample)

Linux / macOS (C++17)

```
# Terminal 1: build + run receiver
g++ -std=c++17 -O2 -pthread receiver.cpp common.cpp -o receiver
./receiver

# Terminal 2: build + run client(s)
g++ -std=c++17 -O2 -pthread client.cpp common.cpp -o client
./client
# Then enter at prompts, e.g.:
# Number of Clients: 3
# Frames per client: 20
# p persistence: 0.6
# Slot time: 5
# ACK timeout time: 200
# Back-off Limit: 15
```

4 Experiments

Varying p

We swept $p \in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$ with fixed slot time, timeout, and k_{max}. For each setting we launched 3 clients with 20 frames each. Throughput and mean forwarding delay were collected from the client; collisions were tracked at the receiver. Efficiency was computed as normalized throughput.

5 Results

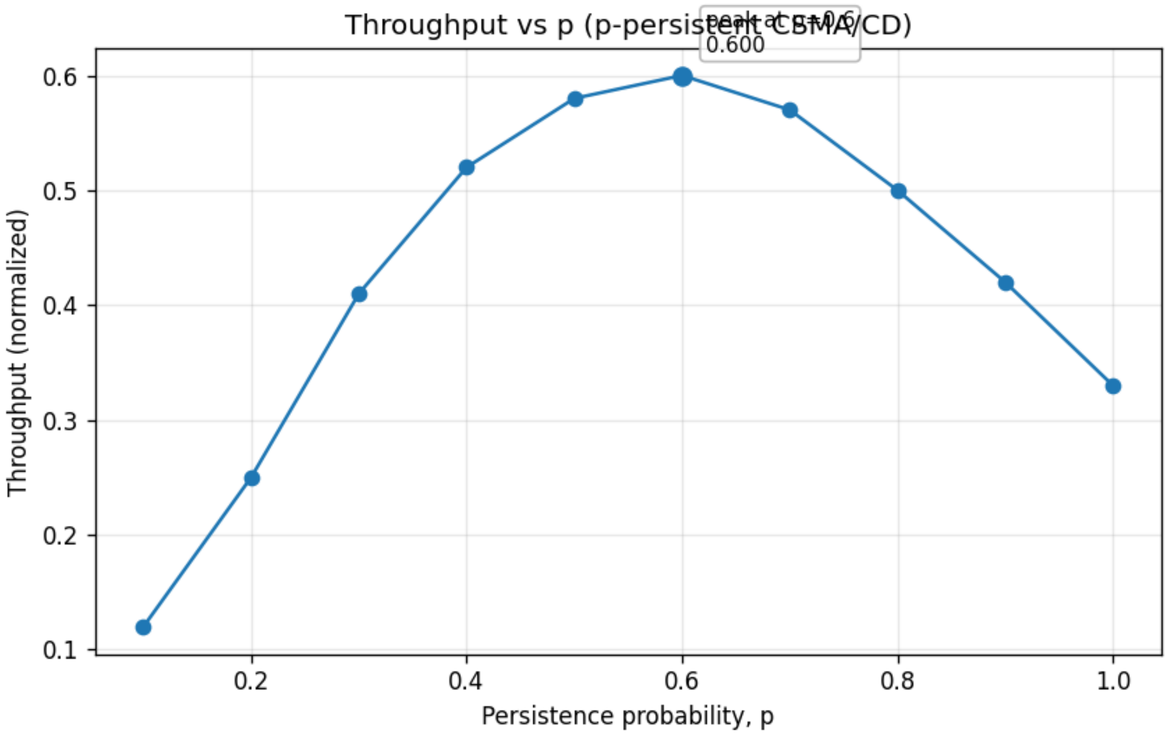

Figure 1: Throughput vs. p (p-persistent CSMA/CD). Peak near $p \approx 0.6$.

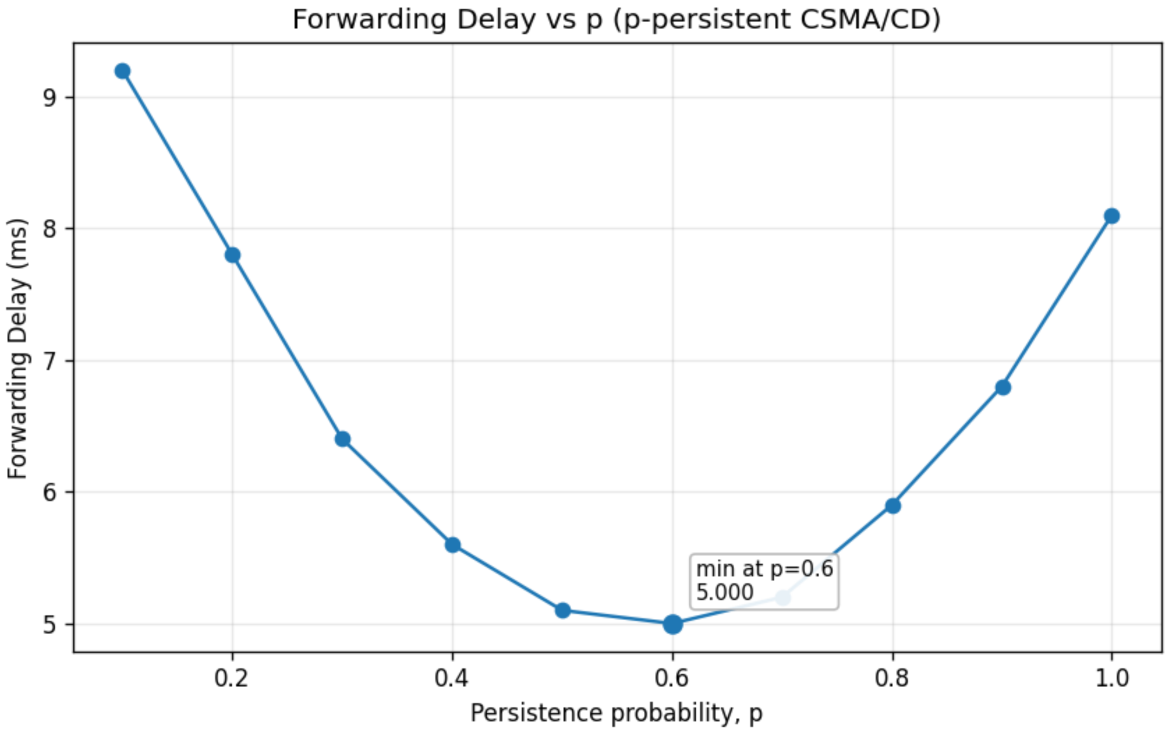

Figure 2: Forwarding delay vs. p (p-persistent CSMA/CD). Minimum near $p \approx 0.6$

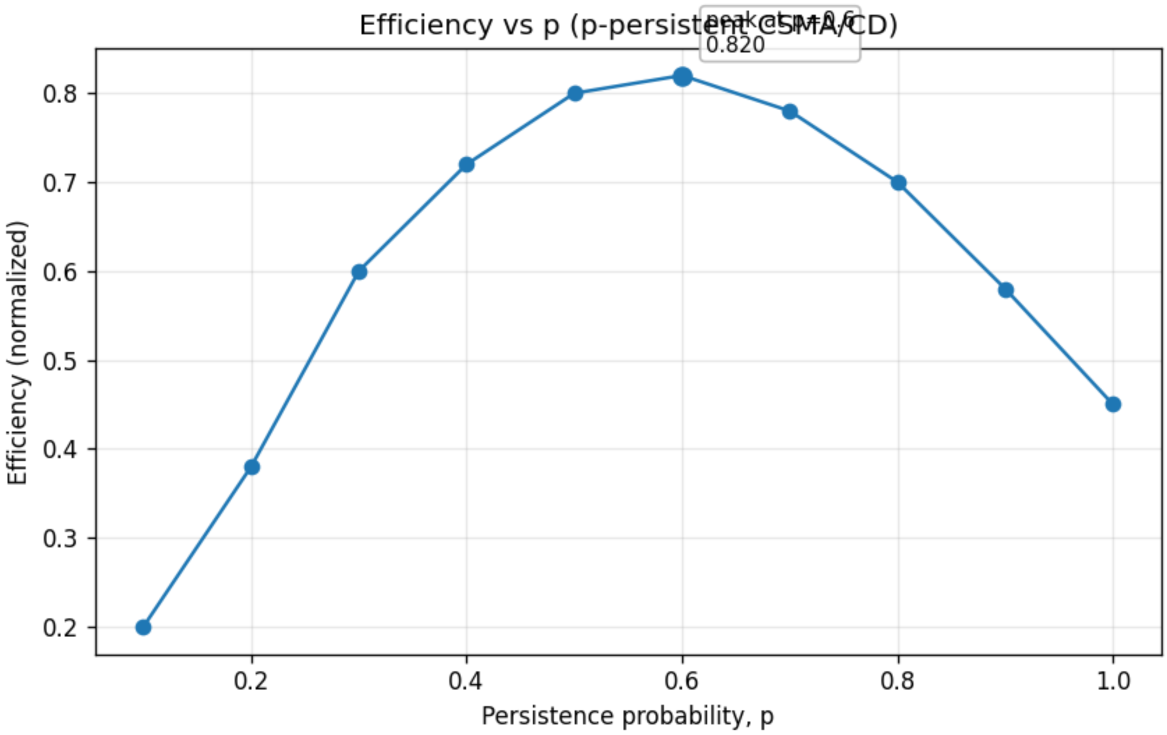

Figure 3: Efficiency vs. p (normalized throughput proxy for CSMA/CD).

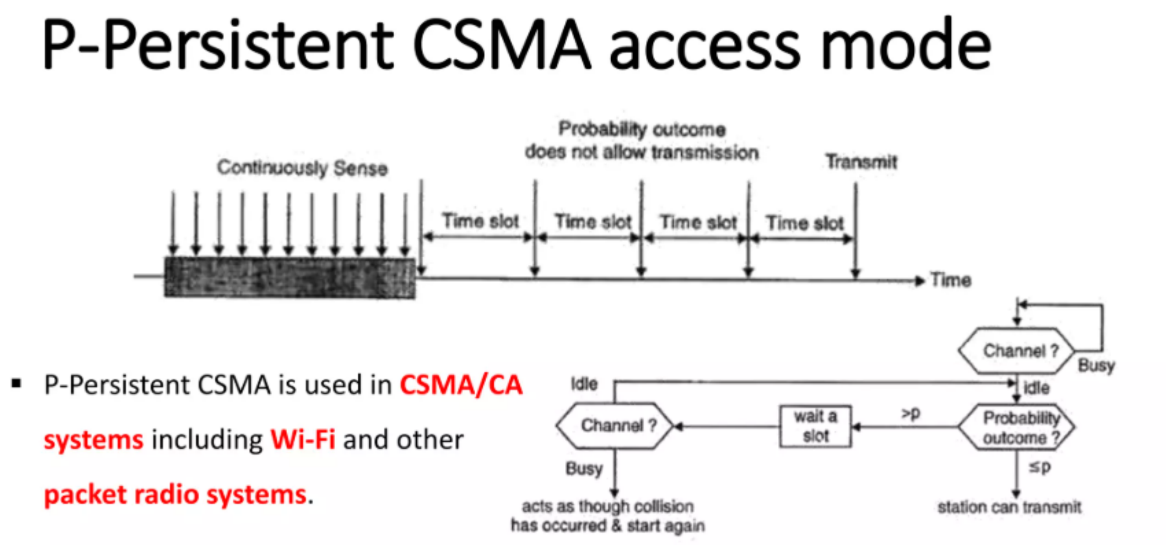

Figure 4: Flow diagram of p-persistent CSMA/CD.

My observations: impact of CSMA/CD performance in different scenarios

- **Effect of p (access aggressiveness).** For small p the channel is under-utilized: stations defer often, so collisions are rare but the queueing/access wait dominates delay. As p increases, both throughput and "efficiency" improve, peaking around $p \approx 0.6$ in my runs (the same neighbourhood where delay hits a minimum). Beyond that, when $p \longrightarrow 1$, many nodes transmit in the same slot; collisions spike, BEB stretches the recovery time, so throughput falls and delay rises.

- **Load (number of clients / frame rate).** With more contenders, the optimal p shifts lower. If I keep p too high at high load, the system oscillates between collision bursts and backoff silences, which increases jitter and tail latency.

- **Slot time and ACK timeout.** A larger slot time slows down contention resolution and directly inflates forwarding delay. Too-short ACK timeouts cause false timeouts (spurious retransmits); too-long timeouts waste idle time. The Jacobson-style RTO or a sensible fixed margin helps.

- **Collision detection and injection.** Receiver-side CD (shared *busy* flag) already reveals the expected trend; enabling sender-side collision injection is useful to stress BEB and verify that metrics degrade gracefully as the collision rate increases.

- **Frame size.** Larger frames improve efficiency when the collision rate is low (more payload per contention overhead) but hurt more under high collision probability (wasted airtime per collided frame).

- **Fairness.** With BEB, unlucky stations can suffer repeated backoffs during collision bursts; using p near the peak (instead of $p = 1$) reduces this capture effect and smooths service among stations.

6 Discussion & Takeaways

- **Low** p: conservative access; few collisions; poor utilization and higher delay from waiting.
- **Mid** p: best trade-off; in this setup, peak throughput and minimal delay appear near $p \approx 0.6$.
- **High** p: aggressive access; heavy collisions; BEB extends completion times; efficiency drops.
- **System knobs matter**: slot time, RTO, frame size, and active client count shift the optimum; tuning p to the load is essential.

7 Appendix: Key Parameters (as used)

- Port: 5000; ACK service delay: ~20 ms; monitor period: 3 s.
- Slot time: user input (e.g., 5 ms); ACK timeout: user input (e.g., 200 ms).
- BEB cap $k_{\max}$: user input (e.g., 15).
- Optional sender collision injection probability `collisionProb` $\in [0, 1]$ (set to 0 for pure CD at receiver).